

Event-driven Adaptation in COP^{*}

Pierpaolo Degano

Gian-Luigi Ferrari

Letterio Galletta

Dipartimento di Informatica, Università di Pisa, Pisa, Italia.

{degano, giangi, galletta}@di.unipi.it

Context-Oriented Programming languages provide us with primitive constructs to adapt program behaviour depending on the evolution of their operational environment, namely the context. In previous work we proposed ML_{CoDa} , a context-oriented language with two-components: a declarative constituent for programming the context and a functional one for computing. This paper describes an extension of ML_{CoDa} to deal with adaptation to unpredictable context changes notified by asynchronous events.

1 Introduction

Software is eating the world: it is a fundamental component of our everyday objects, that all together form what is often called the *Internet of Things*. In this scenario, the physical resources (e.g. your coffemaker) and the virtual ones (e.g. your calendar) create a *highly dynamic and open virtual computing platform*, often referred to as the *context*. The context “virtualises” the things, namely the *smart* resources so as to make them appear less heterogeneous, unlimited and fully dedicated to their users. Also, the context provides an abstract and uniform communication infrastructure to access smart things. A distinguishing characteristic of smart resources is that in principle they are always connected to the Internet, possibly through different access points. Hence, they can interact with each other by collecting and exchanging information of various kinds and perform actions that modify the context. For instance, your smart alarm clock can switch on your coffemaker to prepare you a cup of coffee. In addition, they can choose on their own *where*, *when* and to *whom* they are visible and to *which* portions of the context.

A key challenge is designing software systems that run without compromising their intended behaviour or their non-functional requirements, e.g. quality of service, when injected in such highly dynamic and open contexts. Programming these systems thus requires new programming language features and effective mechanisms to deal with *context-awareness*, i.e. *sensing*, *reacting* and properly *adapting* to changes of the actual context. A comprehensive discussion on the software engineering challenges of the *open world assumption* and of *adaptation* is in [3].

Using traditional programming languages, context-awareness is usually implemented by modelling the context through a special data structure, which can answer to a fixed number of queries, whose results can be tested through suitable **if** statements. This approach however charges the programmer with the responsibility of implementing this data structure and the corresponding operations. In addition, she is in charge of achieving a good modularisation, as well as a good separation of cross-cutting concerns, and of the interactions between the code using the context and legacy code. More flexible proposals are specific design patterns [11] and Aspect-Oriented Programming [8] that encapsulate context-dependent behaviour into separate modules, but still leave most of the burden of handling the context and its changes (and their correctness) to the programmers. Recently, Context Oriented Programming (COP) [6] was proposed as a viable paradigm to develop context-aware software. It advocates languages with peculiar

^{*}Work partly supported by project PRA_2016_64 “Through the fog” funded by the University of Pisa.

constructs that express context-dependent behaviour in a modular manner. In this way context-awareness is built-in, and the provided linguistic abstractions impose a good practice to programmers. This makes a positive impact on the correctness and the modularity of code, mainly because low level details about context management are masked by the compiler.

We proposed in [7] a programming language, called ML_{CoDa} , within the COP paradigm. ML_{CoDa} has two components: a logic constituent for programming the context and a functional one for computing. The logical component provides high level primitives for describing and interacting with complex working environments. Every resource in the context is described by a set of logical predicates representing its state. Thus a programmer can retrieve information from it by asking a query or she can use it by either asserting or retracting logical facts. The functional component offers support for programming a variety of adaptation patterns. Its higher-order facilities are essential to exchange the bundle of functionalities required to manage adaptivity. ML_{CoDa} is equipped with a formal semantics which drove a prototypical implementation in F# [4]. Moreover, ML_{CoDa} offers a further support through a static analysis that guarantees programs to always be able to adapt in every context [7].

Our previous work focussed more on the foundational, linguistic and analysis aspects of context-aware programming, rather than on the way smart resources in the context interact and coordinate. This paper takes a step towards a formal definition of coordination mechanisms. Our context virtualises the smart resources and the communication infrastructure, as well as other software components running within it. Consequently, the dynamic evolution of a context abstractly represents all the interactions of the entities it hosts. It is enough therefore to consider a single application plugged in a context to capture all the interactions with the other entities therein. Our formal coordination model is based on a notion of *event*, thrown to notify when a relevant change occurs in the context. In the wake-up example above, the application is the coffee machine, that will receive an event from the context, actually from the alarm clock, a few minutes before it rings.

Technically, this work enriches the semantics of [7] with mechanisms for throwing, receiving and handling events. Since the context changes independently of the application, events are inherently asynchronous. In our model applications are not strictly event-driven, and execute on their own, provided the resources they need are available. However, the context evolution may make one of these resources unavailable, and upon receiving the related event the application has to appropriately react. Event handling thus deeply affects the run-time behaviour, the formal definition and the implementation of the adaptation constructs. For example, if your application is sending a message through an access point, upon receiving an event of disconnection, it has to adapt its behaviour by connecting to another access point, if any, and resend the undelivered message.

Structure of the paper Section 2 briefly surveys the execution model of ML_{CoDa} and introduces through a running example the main features of our proposal. The operational semantics of our extension is in Section 3, and then we conclude and shortly discuss related and future work in the last section.

2 An example

In this section we illustrate and discuss our proposal, by intuitively presenting the main features of ML_{CoDa} . In the following example, we consider a multimedia guide for museums as an instance of the Internet of Things scenario. We assume the museum has a wireless infrastructure that provides communication facilities, i.e. an Intranet. First a user registers at a desk, specifies her profile and gets credentials, then she connects to the museum Intranet to download the guide for her smartphone.

The museum context As said, the notion of *context*, i.e. the environment where applications and smart objects run, is fundamental for adaptive software. Abstractly, it could be considered as a heterogeneous collection of data coming from different sources and having different representations. In [7] we adopted a declarative approach to deal with the context: it is a knowledge base implemented as a Datalog program, following a well-studied approach [10, 9]. Hence, programmers can concentrate on *what* information the context has to include, leaving to the runtime support *how* this information is actually collected and managed. In practice, an ML_{CoDa} context is a set of facts that predicate over a possibly rich data domain, and a set of logical rules to deduce further implicit properties of the context itself. Adaptive programs can thus query the context and retrieve relevant information, by simply checking a Datalog goal.

Below, we consider a small portion of our guide in order to show how to describe the relevant contextual information declaratively. The following snippet of code declares the pricing policy of the museum, that together with the user profile supports the emission of the appropriate ticket. In our example, the first two clauses grant a reduced ticket to under 10 or over 65 users from a European country; the third clause does the same to students; additionally if they study art, entrance is free:

```
ticket(reduced) ← user_age(x), x < 10, user_country(y), Europe(y).
ticket(reduced) ← user_age(x), x > 65, user_country(y), Europe(y).
ticket(reduced) ← user_work(student).
ticket(free) ← user_work(student), user_study(art).
```

The predicates `user_age`, `user_country`, `user_work` and `user_study` retrieve data from the user profile and we check if they satisfy the given constraints, e.g. `x < 10` or `Europe(y)`.

Adaptation constructs The functional part of the language provides two main mechanisms for programming adaptation. The first is *context-dependent binding* through which a programmer declares variables whose values depend on the context: `dlet x = e1 when goal in e2` means that the variable `x` (called *parameter* hereafter) may denote different objects, with different behaviour depending on the different properties of the current context, expressed by `goal`.

The second mechanism is based on *behavioural variations*, chunks of code that can be activated depending on information picked up from the context, so as to dynamically adapt the running application. It is a list of pairs `goal.expression` within curly parenthesis, similar to pattern-matching, that alters the control flow of applications according to which of its goals holds in the context. Behavioural variations have parameters and are (high-order) values so facilitating programming dynamic adaptation patterns.

Now, we show how behavioural variations express context dependency in our museum guide. Users can buy tickets through their smartphone, either via the museum web page or via a text message. The preferred way is stored in the user's profile, together with additional information (e.g. her nationality). The payment is implemented by the behavioural variation `buy` with parameter `user_id` in the following snippet, where we use a sugared syntax of ML_{CoDa} :

```
fun buyTicket (ticket_kind) =
  let buy = (usr_id){
    ← payByWeb.
    let c = getPage () in
      sendData c ticket_kind usr_id
    ← payByText.
    let c = getNumber () in
      sendText c ticket_kind usr_id
  } in
  #buy (get_usr_id ())
```

The function `buyTicket` takes in input the kind of ticket (previously inferred querying the user profile), and then applies the relevant case of the behavioural variation `buy` to the `usr_id` of the buyer (syntactically, we prefix this kind of application with `#` to distinguish it from the standard functional one). Assume the user pays via a text (the goal $\leftarrow \text{payByText}$ is satisfied), so the second case of the behavioural variation is selected, but just before sending the text (`sendText` invocation), the signal of the mobile network is lost. This change of the context raises the event `signalLost` which is notified to the smartphone. The running application must then adapt his behaviour to the asynchronous notification. In our example, the guide reacts to the event by re-executing the behavioural variation above selecting another alternative to complete the payment, provided that the smartphone can connect to the WiFi.

Of course, the interaction with the Datalog context is not limited to queries, but one can change the knowledge base through **tell/retract** operations that add/remove facts. Modifying the context via these operations could raise specific events for other applications in the context: an event could be abstractly thought as an asynchronous sequence of **tell** and **retract**. In our example, the network failure is caused by another entity in the context through the operation **retract** (`phone_on`).

Note that event-driven adaptation cannot be rendered by using conditional statements alone, since the change (notified by an event) may falsify the guard of the already selected behavioural variation, e.g. because a resource becomes unavailable. Consequently, mechanisms are in order to automatically re-adapt the application to the new context, if possible at all.

We now intuitively illustrate how behavioural variations work and how events are handled influencing adaptation. As we will formalise later on, the semantics of ML_{CoDa} is given through a transition system the configurations of which have the form $\langle q, \rho, C, p \rangle$ where q is an event queue, ρ is an environment for binding parameters, C is (a simplified form of) the context and p the running program; we also assume to have a list of event handlers h . Consider the snippet above and suppose that its execution reaches the point in which the behavioural variation `buy` is applied, represented by the following configuration:

$$\langle \varepsilon, \rho, C, \# \text{buy}(\text{id}) \rangle$$

where ε stands for the empty queue of events, the actual value of ρ is immaterial, and the context C contains the fact `phone_on` and `id` is the value returned by `get_usr_id`. When the behavioural variation is applied, the second case of `buy` is selected as said above. A portion of the current configuration is stored because it can turn out to be useful in restoring the execution after the occurrence of an event. In particular, we store the successful goal of `buy`, the environment ρ and the behavioural variation application itself. After few execution steps we obtain the following configuration:

$$\langle \varepsilon, \rho, C, \text{sendText } c \text{ ticket_kind } \text{usr_id} \rangle$$

As described above the event `signalLost` occurs and the configuration evolves to

$$\langle \text{signalLost}, \rho, C, \text{sendText } c \text{ ticket_kind } \text{usr_id} \rangle$$

The context becomes $C' = C \setminus \{\text{phone_on}\}$ and the event queue is not empty, so we run the handler $e = h(\text{signalLost})$ which connects the smartphone to the WiFi. The new configuration stores in the forth position that the program computation is suspended for the completion of the handler e :

$$\langle \varepsilon, \rho, C', [e] \text{sendText } c \text{ ticket_kind } \text{usr_id} \rangle$$

When the handler terminates we retrieve the information stored when entering the behavioural variation, that is the triple $(\leftarrow \text{payByText}, \rho, \# \text{buy}(\text{id}))$. Since the goal does not hold in C' we apply again the

whole behavioural variation reaching the configuration:

$$\langle \varepsilon, \rho, C', \#buy(id) \rangle$$

and the computation goes on by selecting the first alternative. Instead, if the goal had still held, the computation would have continued by invoking `sendText`.

3 The language

In this section we introduce our formal model by extending the semantics of ML_{CoDa} presented in [7]. As discussed above, the context “virtualises” *smart* objects by providing them with a common interface and a communication infrastructure through which they can interact. Since the interaction is performed by changing the context, in our semantics we only consider the application in hand, its running context and the raised events. The execution of an application and event notifications are asynchronous, and thus the application does not see the context changes caused by an event until it has been fully handled. An application has event handlers for reacting to specific events. The execution of an event handler is atomic, i.e. the application cannot start handling a new event before the running handler has completed.

Syntax The syntax of the Datalog component is standard so here we assume it given [5]. The syntax of the functional part is the following where we denote with F and G Datalog facts and goals, respectively:

$$\begin{aligned} \rho &\in PEnv \quad x, f \in Var \quad \tilde{x} \in DynVar (Var \cap DynVar = \emptyset) \quad \alpha \in Event \\ Va &::= G.e \mid G.e, Va & v &::= c \mid \lambda_f x.e \mid (x)\{Va\} \mid F \\ e &::= v \mid x \mid \tilde{x} \mid e_1 e_2 \mid \mathbf{let} x = e_1 \mathbf{in} e_2 \mid \mathbf{dlet} \tilde{x} = e_1 \mathbf{when} G \mathbf{in} e_2 \mid \\ &\quad \mathbf{if} e_1 \mathbf{then} e_2 \mathbf{else} e_3 \mid \mathbf{tell}(e_1) \mid \mathbf{retract}(e_1) \mid e_1 \cup e_2 \mid \#(e_1, e_2) \mid aux_e \\ h &::= \alpha \Rightarrow e \mid h; h & p &::= [e_1]e_2 \mid e & aux_e &::= e_1^{(G, \rho, e_2)} \mid \bar{e} \end{aligned}$$

It is an extension of the one presented in [7]: values are constants, functions (here written in the standard λ -notation and in the example declared via the keyword **fun**), behavioural variations and facts; expressions include standard ML constructs (**let**, **if**, etc.), context-dependent binding (**dlet**), context updates and behavioural variation application and concatenation. We also have *parameters*, i.e. variables $\tilde{x} \in DynVar$ the value of which can only be known when the running context is fully set up. The novelties of this work are *event handlers* h which manage an event α by running e , and two kinds of extended expressions, p and aux_e , discussed below and used in the semantic definitions.

Semantics For the Datalog evaluation we adopt the top-down standard semantics for stratified programs [5]. Given a context $C \in Context$ and a goal G , we write $C \models G$ with θ to mean that there exists a substitution θ , replacing constants for variables, such that the goal G is satisfied in the context C .

The semantics of ML_{CoDa} is defined by two transition systems working in a master-slave manner. They are inductively defined for expressions with no free variables, but possibly with free parameters, that take a value in an environment $\rho \in PEnv$, updated through the standard operator $\rho[\tilde{x} \mapsto b]$. Actually, we assume that the arrival of an event α transforms a context C in C' , written $C \xrightarrow{\alpha} C'$; here we are not interested in how this happens, so this additional transition system is left abstract.

The master transition system models receiving (i.e. queuing) and handling events. Its transitions have the form $\langle q, \rho, C_\bullet, p \rangle \xrightarrow{h} \langle q', \rho', C'_\bullet, p' \rangle$ where h is a list of event handlers, q is an event queue, ρ is an

environment, C is the context and p the running program. By an abuse of notation, we use h as a function from events to expressions: $h(\alpha) = ()$ when there is no handler for α , otherwise h returns the expression to run. We denote with ε , $\alpha \cdot q$ and $q \cdot \alpha$, resp., the empty queue, a queue with front event α , the queuing of α , resp. The annotation $\bullet \in \{\exists, -\}$ on the context C is a flag used to signal an event notification to the slave transition system; if $\bullet = \exists$, the event may affect the application execution, otherwise it is harmless. The rules of the master transition system are the following:

$$\begin{array}{c}
\text{ENEW} \\
\frac{}{\langle q, \rho, C_\bullet, p \rangle \rightarrow_h \langle q \cdot \alpha, \rho, C_\bullet, p \rangle} \\
\\
\text{EMAN} \\
\frac{}{\langle \alpha \cdot q, \rho, C_\bullet, e \rangle \rightarrow_h \langle q, \rho, C'_\bullet, [e']e \rangle} \quad h(\alpha) = e' \quad C \xrightarrow{\alpha} C' \\
\\
\text{EHDR1} \\
\frac{\langle \rho, C_\bullet, e_1 \rangle \rightarrow \langle \rho', C'_\bullet, e'_1 \rangle}{\langle q, \rho, C_\bullet, [e_1]e_2 \rangle \rightarrow_h \langle q, \rho', C'_\bullet, [e'_1]e_2 \rangle} \\
\\
\text{EHDR2} \\
\frac{}{\langle q, \rho, C_\bullet, [()]e \rangle \rightarrow_h \langle q, \rho, C_\bullet, e \rangle} \\
\\
\text{EEXPR} \\
\frac{\langle \rho, C_\bullet, e \rangle \rightarrow \langle \rho', C'_\bullet, e' \rangle}{\langle \varepsilon, \rho, C_\bullet, e \rangle \rightarrow_h \langle \varepsilon, \rho', C'_\bullet, e' \rangle}
\end{array}$$

The rule ENEW queues a new event α in q , upon arrival; it resembles an early input rule, and it is always enabled. The rule EMAN dequeues an event α from q , updates the context ($C \xrightarrow{\alpha} C'$) and launches the corresponding event handler $h(\alpha)$, recording in the extended expression $[e']e$ both the handler and the suspended application. Since $[e']e$ is not an expression e , we guarantee the atomicity of event handlers. We also assume that event handlers do not perform long run tasks, and that always terminate. The rules EHDR1 and EHDR2 run an event handler (within an extended expression), while the rule EEXPR runs an expression. Note that this rule can only be used when the event queue is empty, ensuring that the application runs in a context which is “stable”.

We now briefly present the rules of the slave transition system, starting from the ones for the usual ML constructs. They are quite standard (just remove the context C_\bullet), so we do not comment on them:

$$\begin{array}{c}
\text{IF1} \\
\frac{\langle \rho, C_\bullet, e_1 \rangle \rightarrow \langle \rho', C'_\bullet, e'_1 \rangle}{\langle \rho, C_\bullet, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rangle \rightarrow \langle \rho', C'_\bullet, \text{if } e'_1 \text{ then } e_2 \text{ else } e_3 \rangle} \\
\\
\text{IF2} \\
\frac{}{\langle \rho, C_\bullet, \text{if true then } e_2 \text{ else } e_3 \rangle \rightarrow \langle \rho, C_\bullet, e_2 \rangle} \\
\\
\text{IF3} \\
\frac{}{\langle \rho, C_\bullet, \text{if false then } e_2 \text{ else } e_3 \rangle \rightarrow \langle \rho, C_\bullet, e_3 \rangle} \\
\\
\text{LET1} \\
\frac{\langle \rho, C_\bullet, e_1 \rangle \rightarrow \langle \rho', C'_\bullet, e'_1 \rangle}{\langle \rho, C_\bullet, \text{let } x = e_1 \text{ in } e_2 \rangle \rightarrow \langle \rho', C'_\bullet, \text{let } x = e'_1 \text{ in } e_2 \rangle} \\
\\
\text{LET2} \\
\frac{}{\langle \rho, C_\bullet, \text{let } x = v \text{ in } e_2 \rangle \rightarrow \langle \rho, C_\bullet, e_2\{v/x\} \rangle} \\
\\
\text{APP1} \\
\frac{\langle \rho, C_\bullet, e_1 \rangle \rightarrow \langle \rho', C'_\bullet, e'_1 \rangle}{\langle \rho, C_\bullet, e_1 e_2 \rangle \rightarrow \langle \rho', C'_\bullet, e'_1 e_2 \rangle} \\
\\
\text{APP2} \\
\frac{\langle \rho, C_\bullet, e_2 \rangle \rightarrow \langle \rho', C'_\bullet, e'_2 \rangle}{\langle \rho, C_\bullet, (\lambda_{fx}.e) e_2 \rangle \rightarrow \langle \rho', C'_\bullet, (\lambda_{fx}.e) e'_2 \rangle} \\
\\
\text{APP3} \\
\frac{}{\langle \rho, C_\bullet, (\lambda_{fx}.e) v \rangle \rightarrow \langle \rho, C_\bullet, e\{v/x, (\lambda_{fx}.e)/f\} \rangle}
\end{array}$$

The rules that handle the context are taken from [7] and are as follows:

$$\begin{array}{c}
\text{TELL1} \\
\frac{\langle \rho, C_\bullet, e \rangle \rightarrow \langle \rho', C'_\bullet, e' \rangle}{\langle \rho, C_\bullet, \text{tell}(e) \rangle \rightarrow \langle \rho', C'_\bullet, \text{tell}(e') \rangle} \\
\\
\text{TELL2} \\
\frac{}{\langle \rho, C_\bullet, \text{tell}(F) \rangle \rightarrow \langle \rho, C_\bullet \cup \{F\}, () \rangle} \\
\\
\text{RETRACT1} \\
\frac{\langle \rho, C_\bullet, e \rangle \rightarrow \langle \rho', C'_\bullet, e' \rangle}{\langle \rho, C_\bullet, \text{retract}(e) \rangle \rightarrow \langle \rho', C'_\bullet, \text{retract}(e') \rangle} \\
\\
\text{RETRACT2} \\
\frac{}{\langle \rho, C_\bullet, \text{retract}(F) \rangle \rightarrow \langle \rho, C_\bullet \setminus \{F\}, () \rangle}
\end{array}$$

$$\begin{array}{c}
 \text{DLET1} \\
 \hline
 \langle \rho, C_{\bullet}, \text{dlet } \tilde{x} = e_1 \text{ when } G_1 \text{ in } e_2 \rangle \rightarrow \langle \rho[\tilde{x} \mapsto G_1.e_1, \rho(\tilde{x})], C_{\bullet}, \overline{e_2} \rangle
 \end{array}
 \qquad
 \begin{array}{c}
 \text{DLET2} \\
 \hline
 \langle \rho, C_{\bullet}, e \rangle \rightarrow \langle \rho', C'_{\bullet}, e' \rangle \\
 \langle \rho, C_{\bullet}, \bar{e} \rangle \rightarrow \langle \rho', C'_{\bullet}, \bar{e}' \rangle
 \end{array}$$

$$\begin{array}{c}
 \text{DLET3} \\
 \hline
 \langle \rho[\tilde{x} \mapsto Va], C_{\bullet}, \bar{v} \rangle \rightarrow \langle \rho, C_{\bullet}, v \rangle
 \end{array}
 \qquad
 \begin{array}{c}
 \text{APPEND1} \\
 \hline
 \langle \rho, C_{\bullet}, e_1 \rangle \rightarrow \langle \rho', C'_{\bullet}, e'_1 \rangle \\
 \langle \rho, C_{\bullet}, e_1 \cup e_2 \rangle \rightarrow \langle \rho', C'_{\bullet}, e'_1 \cup e_2 \rangle
 \end{array}
 \qquad
 \begin{array}{c}
 \text{APPEND2} \\
 \hline
 \langle \rho, C_{\bullet}, e_2 \rangle \rightarrow \langle C'_{\bullet}, e'_2 \rangle \\
 \langle \rho, C_{\bullet}, bv \cup e_2 \rangle \rightarrow \langle \rho', C'_{\bullet}, bv \cup e'_2 \rangle
 \end{array}$$

$$\begin{array}{c}
 \text{APPEND3} \\
 \hline
 \langle \rho, C_{\bullet}, (x)\{Va_1\} \cup (y)\{Va_2\} \rangle \rightarrow \langle \rho, C_{\bullet}, (z)\{Va_1\{z/x\}, Va_2\{z/y\}\} \rangle \quad \text{if } z \text{ fresh}
 \end{array}$$

The application can change the context by asserting or removing facts, as shown by the rule TELL2, after the expression has been reduced first to a fact, by the inductive rule TELL1. The rules DLET deal with context-dependent binding. The rule DLET1 extends the environment ρ by appending $G_1.e_1$ in front of the existing binding for \tilde{x} (e_1 is *not* evaluated until the running context is fully known). In addition, it overlines e_2 , so as to record that it can be evaluated in a context where G_1 holds, under the updated environment. The other two rules are standard: the **dlet** inductively reduces to the plain value eventually computed by \bar{e}_2 .

Finally, the rules for $e_1 \cup e_2$ sequentially evaluate e_1 and e_2 until they reduce to behavioural variations (rules (APPEND1, 2)). Then, they are concatenated together by renaming bound variables to avoid name captures (rule (APPEND3)).

The rules for adaptation are as follows:

$$\begin{array}{c}
 \text{VAAPP1} \\
 \hline
 \langle \rho, C_{\bullet}, e_1 \rangle \rightarrow \langle \rho', C'_{\bullet}, e'_1 \rangle \\
 \langle \rho, C_{\bullet}, \#(e_1, e_2) \rangle \rightarrow \langle \rho', C'_{\bullet}, \#(e'_1, e_2) \rangle
 \end{array}
 \qquad
 \begin{array}{c}
 \text{VAAPP2} \\
 \hline
 \langle \rho, C_{\bullet}, e_2 \rangle \rightarrow \langle \rho', C'_{\bullet}, e'_2 \rangle \\
 \langle \rho, C_{\bullet}, \#(bv, e_2) \rangle \rightarrow \langle \rho', C'_{\bullet}, \#(bv, e'_2) \rangle
 \end{array}$$

$$\begin{array}{c}
 \text{VAAPP3} \\
 \hline
 \text{dsp}(C, Va) = (e', G) \\
 \langle \rho, C_{\bullet}, \#((x)\{Va\}, v) \rangle \rightarrow \langle \rho, C_{\bullet}, e' \{v/x\} \rangle^{(G, \rho, \#((x)\{Va\}, v))}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{DYNVAR} \\
 \hline
 \rho(\tilde{x}) = Va \quad \text{dsp}(C, Va) = (e', G) \\
 \langle \rho, C_{\bullet}, \tilde{x} \rangle \rightarrow \langle \rho, C_{\bullet}, e' \rangle^{(G, \rho, \tilde{x})}
 \end{array}$$

The rules VAAPP handle behavioural variations, that are alike abstractions and thus are applied. The first two rules are standard and the third makes use of the so-called *dispatching* mechanism to select the expression to which the behavioural variation reduces. This inspects Va , i.e. a list of pairs (G, e) , to find the first goal G satisfied by the current context C_{\bullet} , under a substitution θ that binds the variables of G . An adaptation failure occurs if no such goal exists. Otherwise, the result is the pair $(e' = e\theta, G)$ and the behavioural variation reduces to the extended expression $e' \{v/x\} \rangle^{(G, \rho, \#((x)\{Va\}, v))}$. The triple records all the information needed to restart the execution of the variation Va if the occurrence of an event makes the goal G no longer satisfiable. Since goals are used to express those properties that select the appropriate case of behavioural variations, if such a situation arises the computation becomes unreliable, e.g. because some resource disappeared. It is convenient then to restart the whole behavioural variation to recover from a possible failure, without undoing the work done so far. A similar situation arises when a parameter \tilde{x} has to be evaluated. From the variation Va bound to \tilde{x} in ρ , through *dsp* the rule DYNVAR selects an expression to which \tilde{x} reduces, if any, and stores the configuration from which to restart the

computation if an event makes the context unstable. The rules that handle recovery are:

$$\begin{array}{c}
 \text{BRK1} \\
 \frac{\langle \rho, C_-, e \rangle \rightarrow \langle \rho', C'_\bullet, e' \rangle}{\langle \rho, C_-, e^{(G, \rho'', e'')} \rangle \rightarrow \langle \rho', C'_\bullet, e'^{(G, \rho'', e'')} \rangle} \\
 \\
 \text{BRK2} \\
 \frac{}{\langle \rho, C_\bullet, v^{(G, \rho'', e'')} \rangle \rightarrow \langle \rho, C_\bullet, v \rangle} \\
 \\
 \text{BRK3} \\
 \frac{C \models G \quad \langle \rho, C_\exists, e \rangle \rightarrow \langle \rho', C'_\bullet, e' \rangle}{\langle \rho, C_\exists, e^{(G, \rho'', e'')} \rangle \rightarrow \langle \rho', C'_\bullet, e'^{(G, \rho'', e'')} \rangle} \\
 \\
 \text{BRK4} \\
 \frac{C \not\models G}{\langle \rho, C_\exists, e^{(G, \rho'', e'')} \rangle \rightarrow \langle \rho, C_-, e'' \rangle}
 \end{array}$$

The actual recovery is specified by the rule BRK4 that restores the situation *ex quo ante*. Note that the index of C is \exists showing that the event affects the context. Rule BRK1 simply evaluates e , while BRK3 says that the event does not affect the satisfiability of the goal G (note that the index \exists becomes $_$). When a value is reached, annotations are discarded (BRK2).

Discussion Our programming model can be ideally seen as composed by two threads, that operate in an interleaving fashion. The first is the application one, and the other encapsulates, through the context, all the other entities within it. The interleaved execution gives raise to non-determinism, especially because the occurrence of events is asynchronous. Our proposal guarantees that each event is handled atomically: before considering a new event, the execution of the handler of the previous one has to be terminated. We assume to have no control on non-determinism, and thus an application may starve, when it spends all its time in handling events incoming one right after the other. A simple improvement would be assigning a priority to events, having in mind that an over-flood of events is infrequently, especially of those with high priority. Finally, note that here we are not specifying any compensation, as our goal is only to guarantee that the application runs in a context with all the needed features. For taking into account compensations, it will be sufficient to enrich the auxiliary expression *aux_e* and modify and extend accordingly the group of rules (BRK).

4 Concluding remarks

We extended the COP language ML_{CoDa} to manage asynchronous changes of a context caused by the occurrence of events. Some events may drive the execution of a behavioural variation into an unreliable state, e.g. when a device accessed in the current run is switched off. Our semantics supports recovery in these cases. A non trivial effort is needed to extend the static analysis of [7] that guarantees adaptations to be always successful. We plan to carry on this check at run-time, when a behavioural variation is about to run. If successful, such check will avoid at all running a recovery procedure. Also, the ML_{CoDa} prototype will be extended to deal with asynchronous events.

Related work As far as we know a limited number of papers in COP literature has considered event-driven adaptation, among which we briefly mention those with features similar to ours. EventCJ [1] is a Java-based language which combines mechanisms from COP with event based context changes. Differently from our proposal, it provides constructs to declare both the events thrown by an application and the transition rules specifying how to change the context when an event is received. The formal semantics of ContextErlang in [12] describes the behaviour of the constructs for adaptation within a distributed and concurrent framework. The messages of ContextErlang are similar to the our asynchronous event, and so is their handling. The language Flute [2] is designed for programming reactive adaptive software.

Flute constrains the execution of a procedure with certain contextual properties specified by a developer. If any of these properties is no longer satisfied, the execution is stopped, until the property holds again. Stopping the execution is like in our approach when goal of a behavioural variation has been falsified because of a context change. However, here we proposed a different restart mechanism, based on the recovery specified by the rules BRK.

References

- [1] Tomoyuki Aotani, Tetsuo Kamina & Hidehiko Masuhara (2011): *Featherweight EventCJ: a core calculus for a context-oriented language with event-based per-instance layer transition*. COP '11, ACM, New York, NY, USA, pp. 1:1–1:7, doi:10.1145/2068736.2068737.
- [2] Engineer Bainomugisha (2012): *Reactive method dispatch for Context-Oriented Programming*. Ph.D. thesis, Comp. Sci. Dept., Vrije Universiteit Brussel.
- [3] Luciano Baresi, Elisabetta Di Nitto & Carlo Ghezzi (2006): *Toward Open-World Software: issue and Challenges*. Computer 39(10), pp. 36–43, doi:10.1109/MC.2006.362.
- [4] Andrea Canciani, Pierpaolo Degano, Gian-Luigi Ferrari & Letterio Galletta (2015): *A Context-Oriented Extension of F#*. In: *FOCLASA 2015, EPTCS 201*, doi:10.4204/EPTCS.201.2.
- [5] Stefano Ceri, Georg Gottlob & Letizia Tanca (1989): *What You Always Wanted to Know About Datalog (And Never Dared to Ask)*. IEEE Trans. on Knowl. and Data Eng. 1(1), pp. 146–166, doi:10.1109/69.43410.
- [6] Pascal Costanza & Robert Hirschfeld (2005): *Language Constructs for Context-oriented Programming: An Overview of ContextL*. In: *Proceedings of the 2005 Symposium on Dynamic Languages, DLS '05*, ACM, New York, NY, USA, pp. 1–10, doi:10.1145/1146841.1146842.
- [7] Pierpaolo Degano, Gian-Luigi Ferrari & Letterio Galletta (2016): *A Two-Component Language for Adaptation: Design, Semantics, and Program Analysis*. IEEE Transactions on Software Engineering TSE In press, doi:10.1109/TSE.2015.2496941.
- [8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm & WilliamG. Griswold (2001): *An Overview of AspectJ*. In JørgenLindskov Knudsen, editor: *ECOOP 2001 — Object-Oriented Programming, Lecture Notes in Computer Science 2072*, Springer Berlin Heidelberg, pp. 327–354, doi:10.1007/3-540-45337-7_18.
- [9] Seng W. Loke (2004): *Representing and Reasoning with Situations for Context-aware Pervasive Computing: a Logic Programming Perspective*. Knowl. Eng. Rev. 19(3), pp. 213–233, doi:10.1017/S0269888905000263.
- [10] Giorgio Orsi & Letizia Tanca (2011): *Context Modelling and Context-Aware Querying*. In Oege Moor, Georg Gottlob, Tim Furche & Andrew Sellers, editors: *Datalog Reloaded, Lecture Notes in Computer Science 6702*, Springer, pp. 225–244, doi:10.1007/978-3-642-24206-9_13.
- [11] Andres J. Ramirez & Betty H. C. Cheng (2010): *Design Patterns for Developing Dynamically Adaptive Systems*. In: *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '10*, ACM, New York, NY, USA, pp. 49–58, doi:10.1145/1808984.1808990.
- [12] Guido Salvaneschi, Carlo Ghezzi & Matteo Pradella (2015): *ContextErlang: A language for distributed context-aware self-adaptive applications*. Sci. Comput. Program. 102, pp. 20–43, doi:10.1016/j.scico.2014.11.016.